**1. WAPP in C/C++ using MPI to compute the sum of all even numbers in an array using scatter and gather operation.**

#include <mpi.h>

#include <stdio.h>

int is\_even(int x) {

return x % 2 == 0;

}

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Get process ID

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Get total number of processes

const int N = 16; // Total array size (divisible by size)

int data[N], local[N / size], local\_even\_sum = 0;

if (rank == 0) {

// Initialize array at root

printf("Original array:\n");

for (int i = 0; i < N; i++) {

data[i] = i + 1;

printf("%d ", data[i]);

}

printf("\n");

}

// Scatter data from root to all processes

MPI\_Scatter(data, N / size, MPI\_INT, local, N / size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Each process calculates sum of even numbers in its chunk

for (int i = 0; i < N / size; i++) {

if (is\_even(local[i])) {

local\_even\_sum += local[i];

}

}

// Gather local sums at root process

int gathered\_sums[size];

MPI\_Gather(&local\_even\_sum, 1, MPI\_INT, gathered\_sums, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

int total\_even\_sum = 0;

for (int i = 0; i < size; i++) {

total\_even\_sum += gathered\_sums[i];

}

printf("Sum of even numbers: %d\n", total\_even\_sum);

}

MPI\_Finalize();

return 0;

}

✅ Example Output (N = 16)

Original array:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Sum of even numbers: 72

**2 WAPP in C/C++ using MP1 to implement the linear search algorithm.**

#include<stdio.h>

#include<mpi.h>

int main(int argc, char \*\*argv)

{

     MPI\_Init(&argc, &argv);

    int rank, size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    const int N=8;

    int data[8]={2,4,5,6,7,8,12,15};

    int key=12;

    int chunk=N/size;

    int subdata[chunk];

    MPI\_Scatter(data, chunk, MPI\_INT, subdata, chunk, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&key, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    int local\_found=0;

    for(int i=0; i<chunk; i++)

    {

        if(subdata[i]==key)

        {

            printf("Key %d is found at %d by processor %d\n", key, rank\*chunk+1, rank);

            local\_found=1;

        }

    }

    int found=0;

    MPI\_Reduce(&local\_found, &found, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

    if (rank == 0 && found == 0)

    printf("Key %d was not found in any process.\n", key);

    MPI\_Finalize();

    return 0;

}

Compiling

Compilation is OK

Execution ...

Key 12 is found at 7 by processor 3

**3. WAPP in C/C++ using MPI to add two compatible matrices.**

#include <stdio.h>

#include <mpi.h>

#define N 4 // Rows

#define M 4 // Columns

int main(int argc, char \*argv[]) {

int rank, size;

MPI\_Init(&argc, &argv); // Initialize MPI

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Get current process rank

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Get number of processes

if (N % size != 0) {

if (rank == 0)

printf("Matrix row count (%d) must be divisible by number of processes (%d)\n", N, size);

MPI\_Finalize();

return 1;

}

int rows\_per\_proc = N / size;

int A[N][M], B[N][M], C[N][M]; // Full matrices (used only by rank 0)

int local\_A[rows\_per\_proc][M]; // Local submatrices

int local\_B[rows\_per\_proc][M];

int local\_C[rows\_per\_proc][M];

// Initialize matrices A and B at root

if (rank == 0) {

printf("Matrix A:\n");

for (int i = 0; i < N; i++) {

for (int j = 0; j < M; j++) {

A[i][j] = i + j;

B[i][j] = (i + 1) \* (j + 1);

printf("%3d ", A[i][j]);

}

printf("\n");

}

printf("\nMatrix B:\n");

for (int i = 0; i < N; i++) {

for (int j = 0; j < M; j++) {

printf("%3d ", B[i][j]);

}

printf("\n");

}

}

// Scatter rows of A and B to all processes

MPI\_Scatter(A, rows\_per\_proc \* M, MPI\_INT,

local\_A, rows\_per\_proc \* M, MPI\_INT,

0, MPI\_COMM\_WORLD);

MPI\_Scatter(B, rows\_per\_proc \* M, MPI\_INT,

local\_B, rows\_per\_proc \* M, MPI\_INT,

0, MPI\_COMM\_WORLD);

// Local computation: local\_C = local\_A + local\_B

for (int i = 0; i < rows\_per\_proc; i++) {

for (int j = 0; j < M; j++) {

local\_C[i][j] = local\_A[i][j] + local\_B[i][j];

}

}

// Gather local\_C results back to C in root

MPI\_Gather(local\_C, rows\_per\_proc \* M, MPI\_INT,

C, rows\_per\_proc \* M, MPI\_INT,

0, MPI\_COMM\_WORLD);

// Display result matrix at root

if (rank == 0) {

printf("\nMatrix C = A + B:\n");

for (int i = 0; i < N; i++) {

for (int j = 0; j < M; j++) {

printf("%3d ", C[i][j]);

}

printf("\n");

}

}

MPI\_Finalize(); // Finalize MPI

return 0;

}

**4. WAPP in C/C++ using MPI to compute the value of PI.**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

static inline double rand01(unsigned int \*seed)

/\* Fast re‑entrant RNG: returns uniform double in [0,1) \*/

{

return rand\_r(seed) / (double)RAND\_MAX;

}

int main(int argc, char \*argv[])

{

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // my rank

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // #processes

long long N\_total = (argc > 1) ? atoll(argv[1]) : 1000000LL;

/\* Optionally broadcast N\_total from rank 0 so all ranks agree \*/

MPI\_Bcast(&N\_total, 1, MPI\_LONG\_LONG, 0, MPI\_COMM\_WORLD);

/\* Divide work: last rank gets the remainder \*/

long long N\_local = N\_total / size;

if (rank == size - 1) N\_local += N\_total % size;

unsigned int seed = (unsigned int)time(NULL) ^ (rank \* 0x9e3779b9U);

long long local\_hits = 0;

for (long long i = 0; i < N\_local; ++i) {

double x = rand01(&seed);

double y = rand01(&seed);

if (x \* x + y \* y <= 1.0) ++local\_hits;

}

long long total\_hits = 0;

MPI\_Reduce(&local\_hits, &total\_hits, 1, MPI\_LONG\_LONG,

MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

double pi\_est = 4.0 \* (double)total\_hits / (double)N\_total;

printf("π ≈ %.12f (samples = %lld, processes = %d)\n",

pi\_est, N\_total, size);

}

MPI\_Finalize();

return 0;

}

**5. WAPP in C/C++ using MPI to compute the sum of all even numbers in an array using scatter and gather operation.**

#include <stdio.h>

#include <mpi.h>

int is\_even(int x) {

return x % 2 == 0;

}

int main(int argc, char \*\*argv) {

int rank, size;

MPI\_Init(&argc, &argv); // Step 1: Initialize MPI

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Step 2: Get rank

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Step 3: Get size

const int N = 16; // Total number of elements (must be divisible by size)

int data[N];

int local\_data[N / size];

if (rank == 0) {

// Initialize array with some numbers

for (int i = 0; i < N; i++)

data[i] = i + 1; // Example: 1 to 16

}

// Step 4: Scatter the array

MPI\_Scatter(data, N / size, MPI\_INT, local\_data, N / size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Step 5: Compute local sum of even numbers

int local\_even\_sum = 0;

for (int i = 0; i < N / size; i++) {

if (is\_even(local\_data[i]))

local\_even\_sum += local\_data[i];

}

// Step 6: Gather local even sums

int all\_even\_sums[size];

MPI\_Gather(&local\_even\_sum, 1, MPI\_INT, all\_even\_sums, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Step 7: Root calculates final sum

if (rank == 0) {

int total\_even\_sum = 0;

for (int i = 0; i < size; i++)

total\_even\_sum += all\_even\_sums[i];

printf("Total sum of even numbers in the array: %d\n", total\_even\_sum);

}

MPI\_Finalize(); // Step 8: Finalize MPI

return 0;

}

6. WAPP in C/C++ using MPI to compute find the largest element in the array using broadcast and reduction paradigm.

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

int rank, size;

const int N = 16;

int data[N];

int local\_max = -1;

int global\_max;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Step 1: Initialize the array in root process

if (rank == 0) {

printf("Input array: ");

for (int i = 0; i < N; i++) {

data[i] = rand() % 100;

printf("%d ", data[i]);

}

printf("\n");

}

// Step 2: Broadcast the full array to all processes

MPI\_Bcast(data, N, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Step 3: Each process works on a segment

int chunk\_size = N / size;

int start = rank \* chunk\_size;

int end = start + chunk\_size;

local\_max = data[start];

for (int i = start + 1; i < end; i++) {

if (data[i] > local\_max) {

local\_max = data[i];

}

}

// Step 4: Reduce all local maxima to find the global max

MPI\_Reduce(&local\_max, &global\_max, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

// Step 5: Root process prints the result

if (rank == 0) {

printf("Largest element in array: %d\n", global\_max);

}

MPI\_Finalize();

return 0;

}

**7. WAPP in C/C++ using MPI to compute the product of two compatible matrices.**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#define M 4 // Rows of A and C

#define N 4 // Columns of A and Rows of B

#define P 4 // Columns of B and C

int main(int argc, char \*argv[]) {

int rank, size;

int A[M][N], B[N][P], C[M][P];

int local\_A[M/4][N], local\_C[M/4][P];

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (M % size != 0) {

if (rank == 0)

printf("Matrix row size M must be divisible by number of processes.\n");

MPI\_Finalize();

return -1;

}

int rows\_per\_process = M / size;

// Initialize matrices A and B in the root process

if (rank == 0) {

printf("Matrix A:\n");

for (int i = 0; i < M; i++) {

for (int j = 0; j < N; j++) {

A[i][j] = i + j;

printf("%3d ", A[i][j]);

}

printf("\n");

}

printf("Matrix B:\n");

for (int i = 0; i < N; i++) {

for (int j = 0; j < P; j++) {

B[i][j] = i \* j;

printf("%3d ", B[i][j]);

}

printf("\n");

}

}

// Scatter rows of A

MPI\_Scatter(A, rows\_per\_process \* N, MPI\_INT, local\_A, rows\_per\_process \* N, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Broadcast matrix B to all processes

MPI\_Bcast(B, N \* P, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Local computation of matrix multiplication

for (int i = 0; i < rows\_per\_process; i++) {

for (int j = 0; j < P; j++) {

local\_C[i][j] = 0;

for (int k = 0; k < N; k++) {

local\_C[i][j] += local\_A[i][k] \* B[k][j];

}

}

}

// Gather results into matrix C

MPI\_Gather(local\_C, rows\_per\_process \* P, MPI\_INT, C, rows\_per\_process \* P, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Display result in root

if (rank == 0) {

printf("Matrix C (Product):\n");

for (int i = 0; i < M; i++) {

for (int j = 0; j < P; j++) {

printf("%4d ", C[i][j]);

}

printf("\n");

}

}

MPI\_Finalize();

return 0;

}

**8. WAPP in C/C++ using MPI to compute the sum of elements in an array using scatter and reduction operation.**

#include <mpi.h>

#include <stdio.h>

#define N 16 // Total number of elements (should be divisible by number of processes)

int main(int argc, char \*argv[]) {

int rank, size;

int data[N]; // Full array (used only by root)

int local[N]; // Over-allocated to maximum possible size

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (N % size != 0 && rank == 0) {

printf("Error: Array size N = %d is not divisible by number of processes = %d\n", N, size);

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

int local\_n = N / size;

// Initialize array only on root process

if (rank == 0) {

for (int i = 0; i < N; i++) {

data[i] = i + 1; // Example: 1, 2, ..., N

}

}

// Scatter the array to all processes

MPI\_Scatter(

data, // send buffer (root only)

local\_n, // number of elements sent to each process

MPI\_INT,

local, // receive buffer (each process)

local\_n, // number of elements to receive

MPI\_INT,

0, // root

MPI\_COMM\_WORLD

);

// Each process calculates the sum of its local part

int local\_sum = 0;

for (int i = 0; i < local\_n; i++) {

local\_sum += local[i];

}

// Reduce all local sums to a single global sum at root

int global\_sum = 0;

MPI\_Reduce(

&local\_sum, // send buffer

&global\_sum, // receive buffer (only on root)

1,

MPI\_INT,

MPI\_SUM,

0,

MPI\_COMM\_WORLD

);

// Print the result at root

if (rank == 0) {

printf("Sum of array elements = %d\n", global\_sum);

}

MPI\_Finalize(); return 0; }

**9. WAPP in C/C++ using MPI to compute the sum of all prime numbers in an array using broadcast.**

#include <mpi.h>

#include <stdio.h>

#include <math.h>

int is\_prime(int num) {

if (num < 2) return 0;

for (int i = 2; i <= sqrt(num); i++)

if (num % i == 0) return 0;

return 1;

}

int main(int argc, char \*argv[]) {

int rank, size;

const int N = 16;

int data[N] = {5, 7, 8, 9, 11, 13, 15, 17, 4, 6, 19, 23, 28, 29, 31, 33};

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

// Broadcast the entire array to all processes

MPI\_Bcast(data, N, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Determine local portion to process

int local\_sum = 0;

for (int i = rank; i < N; i += size) {

if (is\_prime(data[i]))

local\_sum += data[i];

}

int global\_sum = 0;

MPI\_Reduce(&local\_sum, &global\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0)

printf("Sum of prime numbers = %d\n", global\_sum);

MPI\_Finalize();

return 0;

}

**10. WAPP in C/C++ using MPI to find and print the primes numbers in an array**

#include <mpi.h>

#include <stdio.h>

int is\_prime(int n) {

if (n <= 1) return 0;

for (int i = 2; i \* i <= n; ++i)

if (n % i == 0) return 0;

return 1;

}

int main(int argc, char \*argv[]) {

int rank, size;

const int N = 16;

int data[N] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17};

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

int chunk = N / size;

int local[chunk];

MPI\_Scatter(data, chunk, MPI\_INT, local, chunk, MPI\_INT, 0, MPI\_COMM\_WORLD);

int local\_primes[chunk];

int local\_count = 0;

for (int i = 0; i < chunk; i++) {

if (is\_prime(local[i])) {

local\_primes[local\_count++] = local[i];

}

}

int recv\_counts[size]; // Number of primes from each process

MPI\_Gather(&local\_count, 1, MPI\_INT, recv\_counts, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

int displs[size], total = 0;

if (rank == 0) {

displs[0] = 0;

for (int i = 1; i < size; i++) {

displs[i] = displs[i - 1] + recv\_counts[i - 1];

}

total = displs[size - 1] + recv\_counts[size - 1];

}

int all\_primes[N]; // assuming total primes ≤ N

MPI\_Gatherv(local\_primes, local\_count, MPI\_INT,

all\_primes, recv\_counts, displs, MPI\_INT,

0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("Prime numbers in the array:\n");

for (int i = 0; i < total; i++) {

printf("%d ", all\_primes[i]);

}

printf("\n");

}

MPI\_Finalize();

return 0;

}